Linear Logical Algorithms

Robert J. Simmons and Frank Pfenning

Carnegie Mellon University
{rjsimmon,fp}@cs.cmu.edu

Abstract. Bottom-up logic programming can be used to declaratively specify many algorithms in a succinct and natural way, and McAllester and Ganzinger have shown that it is possible to define a cost semantics that enables reasoning about the running time of algorithms written as inference rules. Previous work with the programming language Lollimon demonstrates the expressive power of logic programming with linear logic in describing algorithms that have imperative elements or that must repeatedly make mutually exclusive choices. In this paper, we identify a bottom-up logic programming language based on linear logic that is amenable to efficient execution and describe a novel cost semantics that can be used for complexity analysis of algorithms expressed in linear logic.

Key words: Bottom-up logic programming, forward reasoning, linear logic, deductive databases, cost semantics, abstract running time

1 Introduction

Logical inference rules are a concise and powerful tool for expressing many algorithms in a declarative way. In the last decade, several lines of work have advanced the argument that it is not only possible but convenient to formally reason about the *running time* of algorithms expressed as inference rules.

Work on this topic can be broadly categorized into two groups: work that takes a language similar to the pure bottom up logic programming language presented by McAllester [1] and automates reasoning about the complexity of algorithms expressed in that language [2, 3], and work aimed at allowing analysis for logic programming languages with richer features [4–6].

This paper falls into the second category; we present a bottom-up logic programming language based on intuitionistic linear logic [7] that cleanly integrates a notion of state transition with the saturating forward reasoning present in bottom-up logic programming. We follow the two-part approach taken by McAllester and Ganzinger in [1, 4, 5]. First, we give the language a dynamic cost semantics called the *abstract running time* that looks at a chain of logical inferences as a computation and defines the cost of that computation, and then we describe an interpreter that can be shown to execute those computations in time proportional to the abstract running time. Both of these concepts are critical – without the interpreter, there is no reason to believe that the notion of abstract

running time is based in reality, and without the definition of abstract running time, reasoning about the complexity of algorithms requires understanding the intricacies of the interpreter's implementation.

We start by briefly describing a pure logic programming language [1] in which various graph algorithms and program analyses can be expressed concisely and executed efficiently. One example is the program in Fig. 1 that computes connectivity over an undirected graph.

$$\frac{\mathsf{edge}(x,y)}{\mathsf{edge}(y,x)}\ r1 \qquad \frac{\mathsf{edge}(x,y)}{\mathsf{path}(x,y)}\ r2 \qquad \frac{\mathsf{edge}(x,y) \quad \mathsf{path}(y,z)}{\mathsf{path}(x,z)}\ r3$$

Fig. 1. A simple pure, bottom-up program for computing graph connectivity.

Given a graph $G = (E, V)$, this algorithm starts with a database that has a fact $\mathsf{edge}(\mathsf{a}, \mathsf{b})$ for every edge $(\mathsf{a}, \mathsf{b}) \in E$. The intended meaning of this program is that $\mathsf{path}(\mathsf{a}, \mathsf{b})$ should hold if and only if there is a path between vertex a and b in graph G. Here, and throughout the paper, we will represent constants as $\mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots$ and variables as $x, y, z, \ldots$, and we will insist that all the terms in our database be *ground*, meaning that they contain no free variables, and that all rules be *range-restricted*, meaning that the variables in the conclusions (below the line) are a subset of the variables in the premises (above the line). This last restriction ensures that the database continues to contain only ground facts as new facts are derived.

In order to calculate the path relation, rules are applied exhaustively in the forward direction until *saturation* is reached; that is, until no possible forward inference can cause us to learn anything new. The *closure* of an initial database Γ under the rules in a program P (written $\mathsf{Clo}_P(\Gamma)$ or just $\mathsf{Clo}(\Gamma)$) is the smallest set containing Γ closed under the rules in P. Unlike Datalog, the language contains function symbols, so the closure may be infinite; however, we are interested only in programs with a finite closure.

The pure bottom-up logic programming language sketched above and described fully in [1] and elsewhere has great expressive power but also some obvious limitations. We will briefly mention related work on efforts related to our own.

Consider the way we encoded the graph G in Fig. 1. The collection of edges was represented not as a matrix or an adjacency list, but merely as a collection of facts – the data structure that we were working over was implicit in the database. This idiom of *database-as-data-structure* is a strength of this declarative style of programming, as details of underlying data structures can be omitted. However, because the notion of database we use is one that incrementally "learns" all derivable facts in an unspecified manner, it is difficult to describe algorithms that have distinct states or phases. Several attempts at addressing this problem

amount to the identification of reasonable forms of locally stratified negation, such as temporal [8] or XY [9] stratification. However, stratified negation cannot easily describe algorithms that must repeatedly take only one of a number of possible steps, and this can make specifying greedy algorithms difficult [6].

Several disconnected lines of research have approached this problem. Greco and Zaniolo describe a variant of Datalog with an intrinsic notion of *choice* that has a semantics based on stable models and can naturally express a number of greedy algorithms [6]. They define an execution model for their system and show a number of complexity results, but they do not give a cost semantics, so all complexity results are based on directly reasoning about the interpreter.

Ganzinger and McAllester [5] do not explicitly consider the applicability of their system to greedy algorithms, but they demonstrate that their system, based on *deletion* of facts and *priorities* on rules, can express many of the same algorithms that motivated Greco and Zaniolo, such as algorithms computing minimum spanning trees and shortest paths. Unfortunately, the expressiveness of their system is hard to determine because they define an unusual notion of deletion that does not have any clear logical justification.

Pfenning and López et al. [10, 11] propose linear logic as a more principled foundation of Ganzinger and McAllester's work. They show that their implementation of a linear logic programming language, Lollimon, is powerful enough to express many of the algorithms shown in Ganzinger and McAllester's previous work. However, they cannot reason about the running time of such algorithms, only their correctness, and complexity results would seem to be very difficult to obtain in a language such as Lollimon that allows for almost arbitrary integration of forward and backward chaining.

The primary contribution of this paper is the presentation of a programming language based on bottom-up reasoning in linear logic – essentially a first-order, Horn-like fragment of Lollimon – that is both useful for the specification of algorithms and in the analysis of their running time. To our knowledge, this is the first such result for a programming language based on linear logic. Section 2 describes the use of first-order linear logic in specifying a number of simple algorithms. Section 3 defines the operational semantics and cost semantics of the language, demonstrates the use of cost semantics in reasoning about complexity, and briefly describes the interpreter that demonstrates that the cost semantics are reasonable. Section 4 concludes and mentions a number of possible extensions to the basic, pure language considered here.

2 Bottom-up Programming in Linear Logic

The pure bottom-up logic programming language introduced in the previous section is built from *atomic propositions* like $\mathsf{nat}(n)$, $\mathsf{edge}(a, b)$, and $\mathsf{path}(v, u)$. These facts represent truth in the usual, mathematical sense - the rule $r2$ in Fig. 1 says that if we know that there is an edge between some vertices a and b, we can also know that there is a path between them. However, after we learn $\mathsf{path}(\mathsf{a}, \mathsf{b})$, we still know $\mathsf{edge}(\mathsf{a}, \mathsf{b})$, because we treat truth as *persistent*.

Linear logic has a notion of persistent truth, but also has a notion of truth that describes the current (and possibly changing) state of the world. We refer to this notion of "truth in the current state of the system" as *ephemeral truth*, and in addition to the persistent atomic propositions that we have previously seen, we introduce ephemeral (or *linear*) atomic propositions that we distinguish from persistent propositions by using an underline: $\underline{\mathsf{linear}}(x)$.

$$\frac{\underline{\mathsf{wins}}(x, n) \quad \underline{\mathsf{wins}}(y, n)}{\underline{\mathsf{wins}}(x, \mathsf{s}(n)) \quad \mathsf{won}(x, y, n)}$$

Fig. 2. A simple linear logic program describing arbitrary single-elimination tournaments.

Rules with ephemeral propositions as premises introduce the possibility of changing the state of the world. The rule given in Fig. 2 describes a single-elimination tournament in which any team can play another team that has the same number of wins. If we have two teams a and c that have both won zero games, we can represent this as the two linear atomic propositions $\underline{\mathsf{wins}}(\mathsf{a}, \mathsf{z})$ and $\underline{\mathsf{wins}}(\mathsf{c}, \mathsf{z})$. These atomic propositions satisfy the two premises of the rule in Fig. 2. If we arbitrarily let $x = \mathsf{c}$ and $y = \mathsf{a}$, treating c as the "winning team," the rule represents the possibility of transitioning from a state where both teams a and c have won zero games and are still in the running to a state where team c has won one game and where team a is out of the running. There is no $\underline{\mathsf{wins}}(y, n)$ in the conclusion because the tournament is single-elimination – after losing, a team cannot play any other teams. Applying this rule requires *consuming* the two linear propositions we had before and replacing them with a single new linear atomic proposition $\underline{\mathsf{wins}}(\mathsf{c}, \mathsf{s}(\mathsf{z}))$. Applying the rule also adds the persistent atomic proposition $\mathsf{won}(\mathsf{c}, \mathsf{a}, \mathsf{z})$ to the database, which represents a persistent record of the fact that c defeated a in round z.

Changes to the state of a system are not necessarily reversible. While we could imagine a backtracking semantics that would eventually consider team a beating team c, or consider them playing other teams in the first round, we instead read rules with linear premises as describing a *committed choice* – once we apply a rule that consumes an ephemeral proposition, we will never consider any other way that proposition could have been consumed. Put another way, while our rules may describe a system that can evolve in many ways from an initial state, when reading our rules as an algorithm, the algorithm will follow *one* particular evolution of that system in a *don't-care* nondeterministic manner.

We can use these ephemeral atomic propositions to support algorithms that require certain actions to happen a fixed number of times, as well as algorithms that require some actions to be mutually exclusive. The example in Fig. 3 is a

linear algorithm to compute a spanning tree of a connected, undirected graph $G = (E, V)$ that has some distinguished vertex $\mathsf{root} \in V$. The input to the algorithm is a persistent atomic proposition $\mathsf{edge}(\mathsf{a}, \mathsf{b})$ for every edge $(\mathsf{a}, \mathsf{b}) \in E$ and a single ephemeral atomic proposition $\underline{\mathsf{vert}}(\mathsf{a})$ for every vertex $\mathsf{a} \in V$. We view the relation tree as a directed subgraph of G where $\mathsf{tree}(\mathsf{a}, \mathsf{b})$ is true iff there is an edge from a to b in the tree.

$$\frac{\mathsf{edge}(x,y)}{\mathsf{edge}(y,x)}\ r1 \qquad \frac{\underline{\mathsf{vert}}(\mathsf{root})}{\mathsf{intree}(\mathsf{root})}\ r2 \qquad \frac{\begin{array}{c}\mathsf{edge}(x,y)\\ \mathsf{intree}(x)\\ \underline{\mathsf{vert}}(y)\end{array}}{\begin{array}{c}\mathsf{tree}(x,y)\\ \mathsf{intree}(y)\end{array}}\ r3$$

Fig. 3. Finding a rooted spanning tree of an undirected graph.

Correctness of this spanning tree algorithm follows from invariants maintained by the rules. Take V' to be the set of all x such that $\mathsf{intree}(x)$ holds, and take E' to be the the set of all ordered pairs (x, y) such that $\mathsf{tree}(x, y)$ holds. We have two state invariants, maintained by rule application:

1. E' is a subgraph of E and a spanning tree over V'.
2. The set V/V' is always the set of variables x' such that $\underline{\mathsf{vert}}(x')$ holds.

The two examples in Fig. 4 are more imperative in nature; both take as inputs some multiset of items represented by linear atomic propositions of the form $\underline{\mathsf{item}}(x)$ and place them into a data structure. The program on the left requires an additional input of the form $\underline{\mathsf{list}}(\mathsf{nil})$ and collects items into a list represented by a structured term, using $x :: l$ as a shorthand for $\mathsf{cons}(x, l)$. The program on the right requires no additional inputs, and collects items into a forest of binary-heap-like trees. Trees are represented as linear atomic propositions of the form $\underline{\mathsf{tree}}(n, t)$, where n is a natural number expressing the depth of the tree and t is a structured term representing the actual tree, a term consisting of an item and a list of subtrees.

$$\frac{\begin{array}{c}\underline{\mathsf{item}}(x)\\ \underline{\mathsf{list}}(l)\end{array}}{\underline{\mathsf{list}}(x :: l)} \qquad \Bigg| \qquad \frac{\underline{\mathsf{item}}(x)}{\underline{\mathsf{tree}}(\mathsf{z}, \mathsf{node}(x, \mathsf{nil}))} \qquad \frac{\begin{array}{c}\underline{\mathsf{tree}}(n, \mathsf{node}(x, ts))\\ \underline{\mathsf{tree}}(n, t)\end{array}}{\underline{\mathsf{tree}}(\mathsf{s}(n), \mathsf{node}(x, t :: ts))}$$

Fig. 4. Arbitrarily collecting items in a list (left) or in a forest of trees (right).

These examples bring up another important property of linear/ephemeral propositions. For the purposes of bottom-up logic programming, deriving a persistent proposition twice is not any different than deriving it once; however, with

linear propositions we are concerned with the *multiplicity* of those propositions: having two copies of $\underline{\mathsf{item}}(\mathsf{a})$ is different than having one. We will ensure that we can unambiguously refer to linear propositions by labelling them uniquely. For instance, the list-collection example on the left side of Fig. 4 could, from the multiset of atomic propositions $\{l_0 : \underline{\mathsf{list}}(\mathsf{nil}), l_1 : \underline{\mathsf{item}}(\mathsf{a}), l_2 : \underline{\mathsf{item}}(\mathsf{a}), l_3 : \underline{\mathsf{item}}(\mathsf{b})\}$, derive $\underline{\mathsf{list}}(\mathsf{a} :: \mathsf{b} :: \mathsf{a} :: \mathsf{nil})$ and $\underline{\mathsf{list}}(\mathsf{a} :: \mathsf{a} :: \mathsf{b} :: \mathsf{nil})$, but not $\underline{\mathsf{list}}(\mathsf{a} :: \mathsf{a} :: \mathsf{a} :: \mathsf{nil})$, because there are only two linear resources $\underline{\mathsf{item}}(\mathsf{a})$ and the derivation of that proposition requires three such resources. Committed choice ensures that we will only compute *one* of the three possible lists, or more generally one of the $!n$ possible lists given n distinct items.

These examples demonstrate the power of linear logic to express algorithms that would be difficult or inelegant to code in a system without linear resources. In the next section, we will make this more formal by defining an operational semantics based on linear logic and a cost semantics that allows us to reason about running time and complexity without knowing the details of an interpreter for the language.

3 Language Semantics

In this section, we will develop the tools for reasoning about the algorithms we began to specify in the previous section. Two of the fundamental properties of an algorithm are its run time behavior, specified by an operational semantics, and its running time, specified by a cost semantics. We will describe both.

We have already presented a number of programs, but we will formally define a program P as a series of rules. Each rule has one or more atomic propositions $A_0, \ldots, A_{n-1}$ as premises and zero or more atomic propositions $C_0, \ldots, C_{m-1}$ as conclusions; for example, in clause $r2$ of Fig. 3, $A_0 = \underline{\mathsf{vert}}(\mathsf{root})$ and $C_0 = \mathsf{intree}(\mathsf{root})$. There are two additional restrictions on the form of rules:

- *Range restriction.* The free variables in the conclusion must be a subset of the free variables in the premises. This ensures that a ground database will remain ground when inference rules are applied.
- *Separation.* The program must consistently identify some propositions as linear and some as persistent; this was indicated before by writing linear propositions as $\underline{\mathsf{prop}}$ and persistent propositions as prop. Separation also requires that in any rule with linear atomic propositions among the conclusions $C_0, \ldots, C_{m-1}$, at least one of the premises $A_0, \ldots, A_{n-1}$ must be a linear atomic proposition. This requirement helps ensure that we will not "flood" the database with unlimited copies of ephemeral propositions, and also allows us to implement the saturation function Clo effectively.

3.1 Operational Semantics

In this section, we describe an operational semantics for the language we have defined, noting that the operational semantics does not make much sense as an

implementation, as it makes transparently bad choices like running saturating forward chaining redundantly. The input is a finite *initial state* $\langle \Gamma_0, \Delta_0 \rangle$ where Γ_0 is a set of persistent propositions and Δ_0 is a set of labeled linear propositions; a *program trace* is a finite list of states $\langle \Gamma_0, \Delta_0 \rangle \ldots \langle \Gamma_m, \Delta_m \rangle$.

For each state $\langle \Gamma_i, \Delta_i \rangle$, the operational semantics calculates the saturated database $\mathsf{Clo}(\Gamma_i)$ of all the persistent atomic facts that are implied by Γ_i in P by exhaustive forward reasoning, not involving linear propositions in any way. Assuming the process of saturated forward inference terminates, the operational semantics picks an arbitrary rule $r \in P$ and a grounding substitution σ (that is, a substitution that maps every free variable x in the rule to a variable-free term t) such that, for each premise A_i of the rule r, $A_i\sigma$ is in $\mathsf{Clo}(\Gamma_i)$ (if it is persistent) or Δ_i (if it is linear). Applying that rule removes one or more linear resources from Δ_i and adds each conclusion $C_i\sigma$ to either Γ_i or Δ_i depending on whether C_i is linear or persistent. This results in a new state $\langle \Gamma_{i+1}, \Delta_{i+1} \rangle$, and the trace is extended. If there is no rule r and substitution σ satisfying the conditions described above, then the trace cannot be extended and is called a *complete* trace.

The operational semantics separates treatment of monotonic deduction that involves only persistent propositions and the committed choice reasoning that involves consuming ephemeral propositions. This distinction will be reflected in the definition of abstract running time, but we can already see that it is reflected in the arguments about the *termination* of algorithms. It was mentioned in Section 1 that we have to give an argument that the closure will be finite, as this is not true in general; we also have to give a termination argument bounding the length of the program trace by bounding the number of possible applications of rules with linear premises.

3.2 Linear Logic

While many details are beyond the scope of this paper, we will sketch the description of the language and operational semantics in terms of intuitionistic linear logic; our system is a fragment of the judgmental reconstruction of first order intuitionistic linear logic described in [7, 12]. The necessary fragment of linear logic is roughly analogous to the Horn fragment of standard intuitionistic logic.

Atomic propositions	A
Basic propositions	$Q ::= A \mid !A$
State propositions	$S ::= Q \mid \mathbf{1} \mid S \otimes S$
State transitions	$R ::= S \mid S \multimap R \mid \forall x.R$
Persistent hypotheses	$\Gamma ::= \cdot \mid \Gamma, R\ \mathit{pers}$
Ephemeral hypotheses	$\Delta ::= \cdot \mid \Delta, R\ \mathit{eph}$

The translation of a rule r with premises $A_0, \ldots, A_{n-1}$ and conclusions $C_0, \ldots, C_{m-1}$ is the persistent proposition

$$r : \forall \mathbf{x_0}.Q_0 \multimap \ldots \multimap \forall \mathbf{x_{n-1}}.Q_{n-1} \multimap (Q'_0 \otimes \ldots \otimes Q'_{m-1})$$

with $Q_i = A_i$ if A_i is an ephemeral atomic proposition and $Q_i = !A_i$ if A_i is a persistent atomic proposition, and similarly for Q'_i and C_i. The "curried" form is intended to clarify that the variables $\mathbf{x_i}$ first occur in the premise Q_i.

Judgments in the sequent calculus presentation of intuitionistic linear logic have the form $\Gamma; \Delta \vdash R\ \mathit{eph}$; we write $R\ \mathit{pers}$ to indicate that R is persistent, and we write $R\ \mathit{eph}$ to indicate that R is ephemeral. We omit writing the translated rules from the program P that are tacitly included in Γ.

The concepts of *polarity* and *focusing* as described in [13] are useful in describing logic programming from a proof theoretic perspective. In particular, focusing allows us to define *derived rules* in linear logic for any formula in the fragment described above. If we have a rule with premises $\underline{\mathsf{a}}$ and b, and with conclusions $\underline{\mathsf{c}}$ and d, we express that rule in linear logic as $\mathsf{a} \multimap !\mathsf{b} \multimap (\mathsf{c} \otimes !\mathsf{d})$. If we treat every atomic proposition as having positive polarity, focusing on the persistent proposition $\mathsf{a} \multimap !\mathsf{b} \multimap (\mathsf{c} \otimes !\mathsf{d})$ gives this derived inference rule:

$$\frac{\Gamma; \cdot \vdash \mathsf{b}\ \mathit{eph} \qquad \Gamma, \mathsf{d}\ \mathit{pers}; \Delta, \mathsf{c}\ \mathit{eph} \vdash \gamma}{\Gamma; \Delta, \mathsf{a}\ \mathit{eph} \vdash \gamma}$$

where γ is an arbitrary conclusion.

What we see from these rules is that our "next state" actually appears in the *premise* of the derived rule; this may seem a bit unnatural, but it is consistent with Lollimon and other linear logic programming languages [11].

We have left out the details that allow us to actually prove the following theorems, but we can still state the soundness and (non-deterministic) completeness of our language with respect to linear logic.

Theorem 1 (Soundness of operational semantics).
For any (separated and range-restricted) program P and for any program trace $\langle \Gamma_0, \Delta_0 \rangle \ldots \langle \Gamma_m, \Delta_m \rangle$, given a sequent of the form $\Gamma_m; \Delta_m \vdash \gamma$ for an arbitrary γ, there exists a derivation of $\Gamma_0; \Delta_0 \vdash \gamma$.

Proof. By induction on the length of the abstract trace. We need a lemma that if $A \in \mathsf{Clo}(\Gamma)$, then $\Gamma; \cdot \vdash A\ \mathit{eph}$.

Theorem 2 (Nondeterministic completeness of operational semantics).

For any (separated and range-restricted) program P, if the sequent $\Gamma_0; \Delta_0 \vdash \gamma$ is derivable using the sequent $\Gamma_m; \Delta_m \vdash \gamma$, where Γ_0, Δ_0, Γ_m, and Δ_m contain only ground, atomic propositions and γ is an arbitrary conclusion, then there exists some program trace $\langle \Gamma_0, \Delta_0 \rangle \ldots \langle \Gamma'_m, \Delta_m \rangle$ where $\mathsf{Clo}(\Gamma_m) = \mathsf{Clo}(\Gamma'_m)$.

Proof. By induction on focused derivations. We need a lemma that if $\Gamma; \cdot \vdash A\ \mathit{eph}$, then $A \in \mathsf{Clo}(\Gamma)$.

Theorem 2 says that if we can "work on the left" in linear logic from a sequent $\Gamma_0; \Delta_0 \vdash \gamma$ to a sequent $\Gamma_m; \Delta_m \vdash \gamma$, then some trace obeying the operational semantics follows an equivalent path; however, because the operational semantics allows an arbitrary choice of which applicable rule with linear premises to apply, a correct implementation of the operational semantics might never take such a path.

3.3 Cost Semantics

We define, following Ganzinger and McAllester [4, 5], a cost semantics called the *abstract running time*. This cost semantics will allow us to reason about algorithms written in this language, such as the ones in in Section 2, without considering the details of the implementation. The abstract running time of a trace $\langle \Gamma_0, \Delta_0 \rangle \dots \langle \Gamma_m, \Delta_m \rangle$ is the sum of four components: $|\Gamma_0| + |\Delta_0| + m + \Phi$. The first two components, $|\Gamma_0|$ and $|\Delta_0|$, are just the number of persistent and linear resources (respectively) given as input. The other two components are m, the number of transitions involving rules with linear premises, and Φ, the number of *unique prefix firings* – a quantity we will now define.

Definition 1 (Prefix firing). *Let $\langle \Gamma_0, \Delta_0 \rangle \dots \langle \Gamma_m, \Delta_m \rangle$ be a program trace of a program P. A* prefix firing *is a triple $\langle r, \sigma, [l_0, \dots, l_{k-1}] \rangle$ such that*

- *There is a rule r in P with premises $A_0, \dots, A_{n-1}$.*
- *The substitution σ assigns a ground term for every free variable in the premises $A_0, \dots, A_{k-1}$.*
- *There is some state $\langle \Gamma_i, \Delta_i \rangle$ where for all $0 \leq j < k$, either $A_j\sigma \in \mathsf{Clo}(\Gamma_i)$ or $l_j : A_j\sigma \in \Delta_i$, and either*
  - *All of $A_0, \dots, A_{k-1}$ are persistent atomic propositions, or else*
  - *$k < n$ and there is* **no** *substitution σ' that assigns the same terms as σ to the free variables in $A_0, \dots, A_{k-1}$ and additionally assigns ground terms to all the free variables in A_k such that $A_k\sigma' \in \mathsf{Clo}(\Gamma_i)$ or such that $l : A_k\sigma' \in \Delta'_i$, where Δ'_i is Δ_i with all the linear propositions labeled $l_0, \dots, l_{k-1}$ removed.*

As mentioned previously, if multiple instances of the ground linear proposition appear in Δ_i, they have distinct labels and can be used to form distinct prefix firings. Because we don't care about labels of persistent atomic propositions, and the definition doesn't use them, we write them as an underscore "_".

The majority of the definition just expresses the fact that the order of premises matters; the last bullet point is the complicated one. It describes the conditions where we can ignore would-be prefix firings that include linear propositions; we can do so if we know that, in every state, we will always be able to expand the prefix firing to a larger one.

3.4 Using the Abstract Running Time

Because the operational semantics is quite nondeterministic, and because our cost semantics depends on the number of steps taken using of rules with linear premises, we can expect reasoning in general about the running time of programs to be undecidable. However, for well-designed programs it is usually still possible to effectively reason about both the number of prefix firings and the length of the program trace in order to get an informative abstract running time. We give an example in this section, and the extended technical report [14] shows a

similar analysis that gives the list collection and heap collection example in Fig. 4 running times in $O(n)$ and $O(n \log n)$, respectively, where n is the number of input items.

We will show that the spanning tree algorithm in Fig. 3 has an abstract running time in $O(|E| + |V|)$, that is, proportional to the number of edges plus the number of vertices. The abstract running time is $|\Gamma_0| + |\Delta_0| + m + \Phi$. It is obvious that $|\Gamma_0| = |E|$ and $|\Delta_0| = |V|$ based on how the problem is set up; also, because every linear transition consumes a linear resource corresponding to some $v \in V$, an abstract trace can have at most $|V|$ transitions, which is to say that m is bounded by $|V|$.

We consider the prefix firings for each of the three rules in turn. Rule $r1$ can have at most $2|E|$ prefix firings, as every edge $(\mathsf{a}, \mathsf{b}) \in E$ leads to two facts: $\mathsf{edge}(\mathsf{a}, \mathsf{b})$ and $\mathsf{edge}(\mathsf{b}, \mathsf{a})$. Rule $r2$ has no prefix firings, as it has one linear proposition that is either there or not. Rule $r3$ can have at most $4|E|$ prefix firings. The first premise $\mathsf{edge}(x, y)$ effectively "grounds" the rest of the premises, leading to $2|E|$ prefix firings of the form $\langle r3, \sigma, [\_]\rangle$, and the final state will include $\mathsf{intree}(\mathsf{a})$ for every vertex a, resulting in at most $2|E|$ prefix firings of the form $\langle r3, \sigma, [\_, \_]\rangle$. However, there are no prefix firings of the form $\langle r3, \sigma, [\_, \_, l]\rangle$, because a prefix firing that covers all the premises does not meet the condition that $k < n$. This gives us an abstract running time bounded by $2|V| + 7|E|$, so the abstract running time is in $O(|E| + |V|)$.

3.5 Implementing the Operational Semantics

This theorem describes the relationship between the operational semantics, the cost semantics, and the interpreter; it is a is a close analogue to the comparable theorem in [5].

Theorem 3. *For any terminating program P, there exists an interpreter running on a RAM machine extended with constant time hash table operations such that for any initial state $\langle \Gamma_0, \Delta_0\rangle$ the interpreter executes a complete trace $\langle \Gamma_0, \Delta_0\rangle, \ldots, \langle \Gamma_m, \Delta_m\rangle$ and returns $\mathsf{Clo}(\Gamma_m)$ and Δ_m in time proportional to the abstract running time of the trace.*

Theorem 3 establishes that reasoning about the behavior of algorithms described in the language we have presented is a three-part process. First, we must demonstrate that, for a given program, $\mathsf{Clo}(\Gamma)$ is always finite that no trace of the operational semantics can have unbounded length. Second, we must give a bound to the abstract running time of *all* possible complete traces in terms of the initial state. Having done so, Theorem 3 ties the knot by ensuring that the implementation will execute one of the possible complete traces and will do so in time proportional to the abstract running time of that trace. Because we have bounded the abstract running time of any arbitrary trace, we know that the trace actually executed by the interpreter has an abstract running time within that bound and is therefore executed in time proportional to that bound.

The interpreter that establishes Theorem 3 is sketched here and described fully in the extended version of this paper [14]. Given a program, we create a

derived program where for each rule r with n premises in the original program, the derived program has $2n$ rules and introduces $2n$ new atomic propositions (referred to as derived propositions), one for each premise A_i and one for each prefix $A_0, \ldots, A_i$. The derived propositions expose variables that are shared between premises of a rule, allowing an index to efficiently discover premises with matching instantiations of those variables. Two work lists (queues) – one dealing with persistent propositions and one dealing with ephemeral propositions – together contain all the immediate consequences of the facts in the index.

The portion of the interpreter dealing with purely persistent propositions is similar to the interpreter in [1]. When a fact is removed from the persistent work queue and added to the index, the index is used to find all immediate consequences of that fact and those already in the index; these consequences are added to the queue. The treatment of linear atomic propositions is novel. The index and linear work queues are allowed to temporarily contain multiple derived propositions that are all consequences of Δ_i, the current multiset of non-derived atomic propositions, even if some cannot simultaneously be consequences of Δ_i because they require consuming the same ephemeral propositions. These ephemeral propositions are only consumed when a rule from the original program is applied, in the process removing all the derived propositions that depended on the consumed propositions.

In order to avoid unnecessarily declaring and then deleting atomic propositions, upon removing an item from the linear work queue the index is recursively used to find the *first* atomic proposition implied by the program and the dequeued proposition, find the first atomic proposition implied by that proposition, and so on. Either this will succeed until we have shown that $A_0, \ldots, A_{n-1}$ are all derivable, in which case we apply that rule, or it will fail, in which case backtracking, depth-first search looks for a different way to fully apply the rule. Each failure corresponds to a prefix that cannot be extended; therefore, each successful search can be charged against the number of linear transitions, and each unsuccessful search can be charged against the number of prefix firings resulting from non-extendable prefixes that include linear propositions.

4 Conclusion and Future Work

We have described a bottom-up logic programming language that has a notion of ephemeral truth as well as the more familiar notion of persistent truth, and we have defined a cost semantics that allows for reasoning about the running time of programs written in this language. The language can be used to express and analyze a number of algorithms that have a notion of stateful change or nondeterministic update, and other algorithms are described in the extended version [14]. Our system is unique among similar work in having a proof-theoretic semantics based on focusing and linear logic. In the future, we are interested in pursuing a number of extensions to the language described here, including priorities similar to those in [5], temporal stratification and stratified negation similar to [8], and a notion of equality to describe algorithms that use union-find.

Acknowledgments. We would like to thank Michael Ashley-Rollman, Dan Licata, and the three anonymous reviewers for their comments on earlier drafts of this paper. This material is based upon work supported under a National Science Foundation Graduate Research Fellowship by the first author.

We wish to dedicate this paper to Harald Ganzinger, with whom the second author discussed some of the core ideas presented here, and whose untimely passing prevented him from participating further in this research.

References

1. McAllester, D.A.: On the complexity analysis of static analyses. J. ACM **49** (2002) 512–537
2. Nielson, F., Nielson, H.R., Seidl, H.: Automatic complexity analysis. In: ESOP '02: Proceedings of the 11th European Symposium on Programming Languages and Systems, London, UK, Springer-Verlag (2002) 243–261
3. Liu, Y.A., Stoller, S.D.: From Datalog rules to efficient programs with time and space guarantees. In: PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming, New York, NY, USA, ACM (2003) 172–183
4. Ganzinger, H., McAllester, D.A.: A new meta-complexity theorem for bottom-up logic programs. In: IJCAR '01: Proceedings of the First International Joint Conference on Automated Reasoning, London, UK, Springer-Verlag (2001) 514–528
5. Ganzinger, H., McAllester, D.A.: Logical algorithms. In: ICLP '02: Proceedings of the 18th International Conference on Logic Programming, London, UK, Springer-Verlag (2002) 209–223
6. Greco, S., Zaniolo, C.: Greedy algorithms in Datalog. Theory Pract. Log. Program. **1** (2001) 381–407
7. Chang, B.Y.E., Chaudhuri, K., Pfenning, F.: A judgmental analysis of linear logic. Technical Report CMU-CS-03-131, Carnegie Mellon University (2003)
8. Nomikos, C., Rondogiannis, P., Gergatsoulis, M.: Temporal stratification tests for linear and branching-time deductive databases. Theor. Comput. Sci. **342** (2005) 382–415
9. Arni, F., Ong, K., Tsur, S., Wang, H., Zaniolo, C.: The Deductive Database System LDL++. Theory Pract. Log. Program. **3** (2003) 61–94
10. Pfenning, F.: Linear logical algorithms. In: Workshop on Programming Logics in memory of Harald Ganzinger, Saarbrücken (2005) Invited talk.
11. López, P., Pfenning, F., Polakow, J., Watkins, K.: Monadic concurrent linear logic programming. In: PPDP '05: Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, New York, NY, USA, ACM (2005) 35–46
12. Chaudhuri, K.: The Focused Inverse Method for Linear Logic. PhD thesis, Carnegie Mellon University (2006)
13. Chaudhuri, K., Pfenning, F., Price, G.: A Logical Characterization of Forward and Backward Chaining in the Inverse Method. In: Automated Reasoning. Volume 4130. Springer Berlin / Heidelberg (2006) 97–111
14. Simmons, R.J., Pfenning, F.: Linear logical algorithms. Available at `http://www.cs.cmu.edu/~rjsimmon/drafts`. To be published as CMU Technical Report CMU-CS-08-104 (2008)